Noname manuscript No.
(will be inserted by the editor)

Formalising the Hybrid ERTMS Level 3 specification in iUML-B and Event-B

Dana Dghaym · Mohammadsadegh Dalvandi · Michael Poppleton · Colin Snook

Received: date / Accepted: date

Abstract We demonstrate refinement-based formal development of the hybrid, 'fixed virtual block' approach to train movement control for the emerging *European Rail Traffic Management System* (ERTMS) level 3. Our approach uses iUML-B diagrams as a front end to the Event-B modelling language. We use abstraction to verify the principle of movement authority before gradually developing the details of the *Virtual Block Detector* (VBD) component in subsequent refinements, thus verifying that it preserves the safety properties. We animate the refined models to demonstrate their validity using the scenarios from the *Hybrid ERTMS Level 3* (HLIII) specification. We reflect on our team-based approach to finding useful modelling abstractions and demonstrate a systematic modelling method based on the state and class diagrams of iUML-B. The component and control flow architectures of the application, its environment and interacting systems emerge through the layered refinement process. The runtime semantics of the specification's state-machine behaviour are modelled in the final refinements. We discuss how the model could be used to generate an implementation using code generation tools and techniques.

Keywords ERTMS · Event-B · iUML-B · refinement · validation

All data supporting this study are openly available from the University of Southampton repository at https://doi.org/10.5258/SOTON/D0991.

D. Dghaym · M. Poppleton · C. Snook
ECS, University of Southampton, U.K.
{dd4g12,mrp,cfs}@ecs.soton.ac.uk

M. Dalvandi
Department of Computer Science, University of Surrey, U.K.
m.dalvandi@surrey.ac.uk

A list of abbreviations is provided at the end of the paper.

1 Introduction

The *Hybrid ERTMS Level 3* (HLIII) specification [1] concerns the control of trains moving on a linear track and communicating by radio and trackside equipment. A train movement controller called the *Radio Block Centre* (RBC) manages the *Movement Authority* (MA) granted to each train in mission. The focus of this work, called the *Virtual Block Detector* (VBD), conservatively estimates train locations to a finer granularity than physically detected track sections, and thus reports free virtual track sub-sections available for train movement. Trains and trackside report location data to the VBD. In turn the VBD reports free track sections to RBC. The MA granted to each train consists of a set of sections that the train is permitted to move into. An MA can be *extended* by adding further contiguous sections, or *trimmed* by removing sections that the train has passed through. A *controlled* train is instructed that the sections in its MA are free, allowing it to travel at full speed. A *trusted* train is given a special *on-sight* MA allowing it to move cautiously through sections that are not guaranteed to be free; the driver is trusted to avoid collisions. The key safety property which we verify is that controlled trains do not run into trains that are ahead of them.

The refinement-based Event-B modelling method [2] is an appropriate choice since it allows us to verify key properties while leaving certain features, and interacting components, abstract and underspecified. The architecture can be layered through the refinement: each layer can focus on an abstract component interface, the

environment, or a specific feature of the target system. In their evaluation of seven competing state-based formal methods, [3] judge Event-B favourably, and among the best for verification and tool support. They also find Event-B, while less expressive in refinement modelling than ASMs [4] and Z [5], to be superior "when it comes to (mechanically) proving and verifying refinement", ie in proof support. The Rodin [6] toolkit includes theorem provers and model-checkers. Integration of diagrammatic UML modelling with Formal Methods is well established, e.g. [7,8]. Rodin plugins include diagrammatic modelling notations and tools; we use UML-like iUML-B class diagrams and state-machines [9,10,11]. iUML-B leads to a readable formal specification which is easier for domain experts to validate [12,13]. For validation of scenarios we used the ProB [14] model checker with BMotionStudio [15] visualisation.

Formal models are often presented as if they were developed in perfect inexorable steps when, in practice, they never are. We give an overview of our informal team-based process illustrating the iterations that involved many misunderstandings, failures and rework. The team consisted of research and academic staff who had some experience of formal modelling of railway applications such as interlockings and crossings, but no previous experience of communications-based, virtual section train control. Although there were frequent and extensive discussion between the three original team members (Dghaym, Poppleton and Snook), the main model entry and validation and verification (V&V) activities were carried out by one team member (Dghaym). Improved team-working facilities for Event-B are currently under development to alleviate this. One team member (Dalvandi) was added for specialist advice on developing an implementation from the model.

An early version of this work was published in [16]. The additional contributions presented here are as follows.

- Our previous models have been revised to reflect the new version of the specification [1].
- The model is refined to cover more operational details of the system.
- The model has now been validated using the scenarios from the specification.
- We propose methods to generate an implementation from the model.
- We reflect on the process in a more accessible way.

The specification was re-issued partly to take into account the findings of our earlier work [16] and this was reported as an impact of the Enable-S3 project. Specifically, the problematic revoking of an MA has been removed from the specification although the possibility of an already allocated track section changing state still remains.

Structure The paper is structured as follows. Sec. 2 describes our requirements and modelling strategy, giving a summary of the methods and process we used and recalling Event-B and iUML-B basics. Sec. 3 describes the results of our analysis of the system to derive and clarify the requirements. Sec. 4 presents the details of our modelling including refinement strategy (Sec. 4.1) followed by a detailed account of modelling (Sec. 4.2). Sec. 5 describes our verification by theorem proof (Sec. 5.1) and validation by scenario animation (Sec. 5.2). Sec. 6 discusses ways to derive an implementation from the model. Sec. 7 provides observations about the specification and reflections on the modelling methods arising from the development. Sec. 8 compares our development with other ABZ 2018 publications about the same case study as well as some other related work. Sec. 9 concludes by reviewing the work, its increment over [16] and required future work and tool improvements.

2 Requirements & Modelling Strategy

The HLIII specification focusses on the operational details of the VBD component whereas a formalisation of safety prooperties requires an abstract system level description. Therefore our first stage was to derive requirements via a systems analysis which is described in Section 3. We then proceeded to model the system starting with the *environment* (ENV) and specifying what we mean by safety before introducing the concept of movement authority from the RBC and then VBD operation to ensure safety. Further details of the model structure are given in Section 4. We have not provided traceability from requirements to the formalisation. This would be useful further work. The primary aim and result of our solution is that the safety properties are clearly formalised and the principle of movement authority is proven to be safe (with some caveats) before the operational details of the VBD are proven to maintain this safety while also being demonstrated to satisfy the scenarios given in the specification (again with some caveats). We modelled the requirements fully except that we abstracted away from length measurements such as 'minimum safe distance'. This affects the accuracy of one transition in the operational state-machine. We also abstracted away from an interval measurement of time and therefore do not incorporate any notion of clock in our model. Although the specification involves a number of time deadlines, each one is a simple expiry event which can be fully

modelled via arbitrary expiry events without a measure of time. The remainder of this section summarises the steps of the process that we followed in developing the model.

Systems Analysis While the HLIII specification [1, 17] is well presented in terms of explanatory scenarios, its focus makes it a detailed requirements specification for the VBD. It does not explain the overall system aims and principles so well. We therefore started by reverse engineering our understanding of the system in order to understand its purpose and the concepts on which it is based. This involved analysis of the information in the specification, discussions and sketching whiteboard diagrams such as components, entity relationship and state-machine diagrams. The diagram-based analysis naturally led into the iUML-B modelling. The systems analysis identifies the main components in the system and the information flow between them. This is necessary for the model to reflect the appropriate responsibilities of the VBD versus assumptions it makes upon other components. As with most stages of the modelling process, the analysis was iterative. The modelling improved our understanding of the system and our new understanding helped us choose better abstractions for modelling. For example, initially we assumed that only connected trains were in mission. However, when modelling we realised that when a connection is lost the system relies on the fact that the train will continue to respect its MA and this implies that the train is still in mission. This new understanding of the system led us to revise our models so that the 'in-mission' state-machine was independent of (i.e. parallel with) the connected state-machine.

Refinement Strategy The refinement strategy provides a plan for building the model; choosing abstractions, adding details in refinement steps and introducing invariant properties at appropriate stages. We considered two alternative approaches, a) start from an abstract safe system or b) start from an unsafe system and make it safe. In this work we chose the second approach. While the first approach is perhaps more traditional, in this case, the safety properties were not so obvious and were complicated by unsafe, albeit mitigated, scenarios. So we wanted to capture the essence of train movement before introducing assumptions and progressing towards details that can distinguish between safe scenarios and mitigated unsafe scenarios. Again, the refinement strategy evolved as we discovered difficulties and adapted our approach.

Modelling with Event-B and iUML-B Event-B [2, 18] is a refinement-based formal method for system development. An Event-B model contains two parts: *contexts* for static data, and *machines* for dynamic behaviour specified by *variables* `v`, *invariant* predicates `I(v)` that constrain the variables, and *events*. An event comprises a guard denoting its enabling condition and an action describing how the variables are modified when the event is executed. In general, an event `e` takes the form of the the following definition, where `t` are the event parameters, `G(t, v)` is the guard of the event, and `v := E(t, v)` is the action of the event.

```
e == any t where G(t,v) then v := E(t,v) end
```

Event-B is supported by the *Rodin Platform* [6], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

iUML-B [11,10,9] provides a diagrammatic modelling notation for Event-B in the form of state-machines and class diagrams. The diagrammatic elements share the repository of an Event-B model, and contribute to that model. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states, and transitions contribute additional guards and actions to existing events. Class diagrams provide a way to visually model data relationships providing an object-oriented style 'lifting' which is absent from standard Event-B. Classes, attributes and associations are linked to Event-B data elements (carrier sets, constants, or variables) and generate constraints on those elements while class methods contribute additional guards and actions to existing events. Note that iUML-B is designed to be a diagrammatic representation of Event-B and is syntactically and semantically different from UML. For example, there is no notion of triggers or 'run to completion' in iUML-B, transitions are enabled and may fire when their source state is active and their guard is true. Where several transitions and/or methods are linked (i.e. contribute) to the same event, they must all fire together. This gives a way to synchronise transitions in different state-machines.

For modelling we used iUML-B for its diagrammatic notation which follows on from the diagrams used in our analysis and review stages. We used automatic theorem provers to verify our Event-B models and the ProB model checker to find counter examples when the theorem provers could not discharge proof obligations.

Review We held regular reviews to discuss problems with the modelling. As indicated in the previous steps, the reviews led to significant iterations to our understanding of the system, revisions to our refinement plan

and consequent changes to the model. Problems fell into the following categories:

- We cannot prove this proof obligation (PO) - look for a better modelling approach. Example: Contiguity of *next_VSS* relationship. We found it difficult to prove contiguity properties about *Virtual Sub-Section* (VSS) using properties of the sequence, the obvious Abstract Data Type model; [19] discusses some of the difficulties of modelling with such ADTs. To avoid such effort we used numeric indexing of VSS, thus relying on the contiguity of a range of integers. We retained the *next* function for elegance of expression in guards and actions. Fig. 2a of Sec. 4.2.1 presents the VSS class properties.
- This is not a useful refinement - change refinement strategy. Example: We wished to introduce features such as timers as soon as it was possible to do so (i.e. when the triggering functionality was available). However, we had not yet introduced the relevant VSS state changes to utilise the timeout. To rectify this we altered our refinement strategy to introduce abstract versions of VSS states and associated transitions before introducing timers. This is explained in more detail in Sec. 4.2.2.
- This is not a true data refinement - change systems analysis. Example: As we modelled the flow of information through the control components we found it difficult to reconcile the reported train positions and controlled MA with the safety properties of the abstract environment. It seemed that we would need to introduce some form of responsiveness assumptions to limit the difference between actual and control variables. However, the specification implied that the VSS states were asynchronously updated. As our understanding of the MA principle improved we realised that the position inaccuracy is of no consequence and we adjusted our systems description accordingly.

Validation Once the model had been refined to include the operational details of the VBD, the model was animated to validate it against the example scenarios given in the specification. For this validation stage we created a BMotionStudio visualisation of the railway system and developed a new 'scenario checker' plug-in that automatically animates internal processes of the model and records/replays scenarios involving the external interfaces.

3 Requirements resulting from System Analysis

The HLIII specification is a detailed description of one component (the VBD) of a wider system that controls train movements. The other components involved in the system are the trains and trackside equipment, which we refer to as ENV, and the RBC that calculates movement authorities limiting the movement of trains.

The VBD receives messages from trains and train detectors. It also receives information about the output of the RBC. It calculates a range of sections that it believes to be free of any trains and sends these to the RBC. The RBC sends to each train, a movement authority consisting of a range of sections that the train may move into. The train is either instructed that the sections are all free or that they might not be free. We wish to model and verify item 3, the VBD. To do this we also need to consider (and model) the other 2 items.

The **environment** consists of a linear track divided into fixed sections (*Virtual Sub-Section* (VSS)) with trains moving in one direction on the track. Detectors (*Trackside Train Detection* (TTD)) report when a train is present. However, there is only one TTD for a range of VSS. There are 2 kinds of trains; those that communicate with the control system, and those that do not. Trains that communicate send three items of information to the VBD:

- their current position (more precisely than track sections),
- the length of the train,
- whether the train is confirmed as integral (i.e. the carriages are still all connected together).

Communicating trains are able to receive information about the range of sections they are allowed to move through and whether the authorised track is guaranteed to be free (full-supervision) or not (on-sight). For the purpose of this description we partition trains into three kinds: ghost trains (that are not communicating), controlled trains (that are communicating and the control system authorises to move through sections of track which it guarantees to be vacant), and trusted[1] trains (that are communicating and the control system authorises to move through sections of track which may be occupied). Trains that do not communicate can only be detected by TTD and may move freely subject to certain assumptions concerning physical limitations and those imposed by train design regulations.

The **RBC** grants movement authority (permissions) to the communicating trains. The RBC uses information it receives from the VBD about which VSS are free. An MA consists of a range of track sections that the train is allowed to move through. Under RBC control an MA is dynamically extended from the front VSS, and trimmed from the rear VSS. The train is also in-

[1] *Controlled* and *trusted* (trains) are terms that we have introduced, they are not terms from the specification.

structed as to whether it needs to be responsible for avoiding collisions with trains in front (*On-Sight Movement Authority* (OSMA)) or whether it can assume the track sections are free (*Full Supervision Movement Authority* (FSMA)). Note that the specification makes the assumption that OSMA is safe. It is not within the scope of this study to consider whether this is justified. We assume the RBC always issues safe FSMA in accordance with the information it receives from the VBD. That is, all sections in an FSMA are ones that the VBD has calculated to be free.

The **VBD** is responsible for deciding which VSS are free based on information it receives from the TTD and from *Positive Train Detection* (PTD) communications received from communicating trains. It sends information about which VSS it believes are free to the RBC. Since PTD reports may be intermittent or interrupted and some trains do not communicate at all, the estimate of free VSS is cautious in these circumstances.

The positions of trains that are communicating are known fairly accurately (subject to some lag in communications) from the PTD data sent by the train (position, length and integrity) as well as physical limits on possible train movement in between communications. The position of the train is defined by the range of sections from that occupied by the rear to that occupied by the front; this may be a single section. Some robustness is necessary to accommodate limitations of the communication mechanisms such as temporary loss of communication etc.

The position of a train that is not communicating (i.e. a ghost train) is difficult to determine. This is estimated as a range of sections based on the following:

- its last known position (from a PTD or a loss of integrity),
- how far it could possibly have travelled since its position was known,
- information from trains and free TTD that delimits its movement range.

A ghost train is created in the VBD by one of the following means: a communicating train stops communicating, a TTD spontaneously and unexpectedly detects a train, or a communicating train reports that it has lost integrity.

For loss of integrity, a ghost train is created just behind the communicating train to represent the detached section of carriages. A communicating train is converted to a ghost train if the train's mute timer expires (after communication is lost) or if it sends a mission-end message and terminates communication. A ghost train is removed (i.e. destroyed) by sweeping. Sweeping is the movement of a trusted train (with OSMA) through the sections where the ghost train may be. If the trusted train is able to pass through these sections, the ghost train is judged not to exist. A ghost train may also be converted to a communicating train if it starts communicating with the VBD (either by sending a mission start communication or by re-starting previous communication).

4 Model Details

Our formalisation of the requirements uses iUML-B which imparts a modelling style into the generated Event-B. Our strategy for developing the refinements is shown in Section 4.1 Further details of the strategy and modelling style are shown in Section 4.2 which illustrates how we modelled a) the ENV (Sec. 4.2.1), b) the RBC and c) the VBD (Sec. 4.2.2).

4.1 Refinement Strategy

The model consists of an abstract level and ten refinements, with an additional extension refinement for scenario validations using ProB and BMotionStudio. Each refinement introduces more details about the behaviour. In most cases this is done by superposition, where new data and associated behaviour is added without changing that of the previous level. In some cases, a data refinement is performed where some variables of the previous level are replaced and an invariant gives the correspondence between the state of the old and new data. Both superposition and data refinement refine the behaviour of the system since the behaviour is modelled based on the new data.

ENV-M00 Trains: This is the abstract level of the model. It defines a linked list of trains to keep track of train order and prevent overtaking. Trains are created at the rear of the linked list and removed from its front. We also allow adding a new train in the middle of the linked list as a result of train split.

ENV-M0 Train movement, VSS: Introduces the train movement in terms of VSS section updates, where a VSS section is either free or occupied by a train. The train movement is modelled as an update of the position of either the train front or the rear.

ENV-M1 Ghost vs connected trains: Distinction between connected and ghost (i.e. non-connected) trains, where all new trains join as ghost.

ENV-M2 TTD: Introduces TTD sections which can be either free (no train on any of its VSS) or occupied (a train on at least one of its VSS). The TTD state is immediately updated by train movement events.

RBC-M3 RBC: RBC can grant trains MA. We call a train with MA inMission, where the RBC may extend its MA while connected.

VBD-M4 Position reporting: Introduces the reported versus actual train position with the associated MA trimming. We also introduce the memorised location of a train resulting from communication loss.

VBD-M5 Controlled vs trusted trains: Introduces the concept of available VSS as an abstraction to the detailed VSS state transitions table presented in [1]. Fully supervised FS (controlled) vs on-sight OS (trusted) trains are also introduced. An OS train has unsafe MA (i.e. can include not available VSS) but is assumed not to crash into the back of other trains. An FS train has safe MA and therefore cannot crash into the back of other trains.

VBD-M6 Integrity loss: If a train reports integrity loss as a result of either train split or changed length, the train is considered non-integral.

VBD-M7 Waiting timers: Defines three different states for timers: *Idle*, *Started* or *Expired*. Waiting timers apart from the shadow train timer B are introduced at this level.

VBD-M8 VSS State-machine: Distinguishes three different types of trigger that initiate the VSS state-machine: TTD information, position reports (PTD) and timer expiry. The abstract VSS state-machine is refined to the four states: *free*, *unknown*, *ambiguous* and *occupied*. The propagation timers and any remaining timers are modelled at this level.

VBD-M9 Lower levels: Full VSS state transition as per specification, where transitions from one state to the other are refined to model the different alternatives.

4.2 Modelling

The model consists mainly of three parts: the ENV, RBC and VBD. Compared to our HLIII model presented in [16], the ENV part has not changed much. Since we abstract away the details of *how* the RBC calculates a movement authority, it is dealt with in a single refinement which is not changed in this version. The VBD is modified to conform to the new version of the specification document [1] and to complete the modelling of the VSS operational state-machine.

4.2.1 Modelling the Environment

In the first refinements we focus on modelling the ENV and the possible trackside events, such as train movement, splitting and loss of communication.

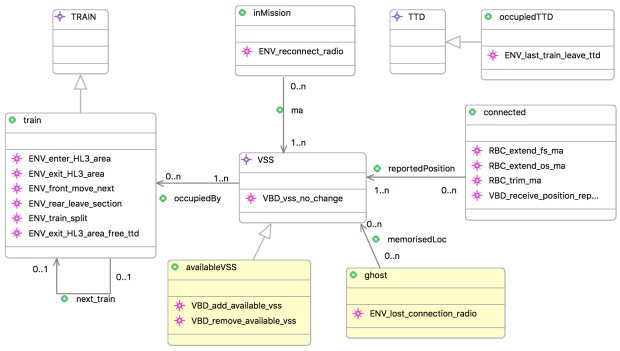


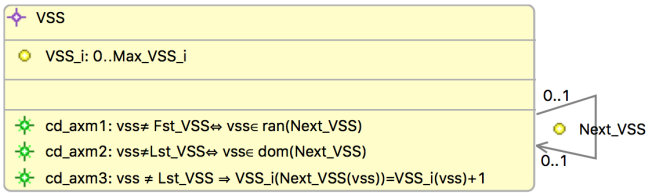
Fig. 1: Class diagram representing dynamic aspects of the environment

Entering & Leaving HLIII Area: In the previous version of the specification [17], the entry/exit of a HLIII area was not mentioned. In our ABZ paper [16], at the abstract level we introduced how trains can join and leave the network. This is similar to the updated specification [1], which includes a new section to describe how trains can enter/exit a HLIII area. Compared to [16], we only updated the event names to conform to the specification.

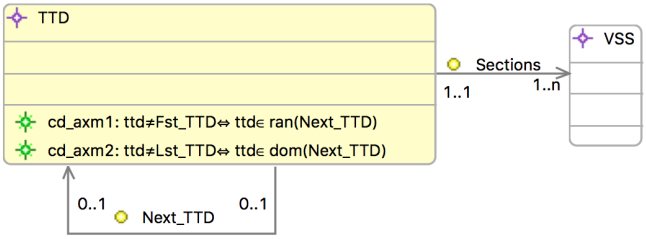
In the model shown in Fig. 1, the variable (green icon) class *train*, with superset *TRAIN* (purple star icon indicating a carrier set), represents the trains that currently exist in the HLIII area. There are two cases for entering an HLIII area, i.e. for creating trains: either a train can join from the beginning of the network (method *Env_enter_HL3_area* in class *train*) , or in the middle as a result of splitting behind an existing train (method *Env_train_split* in class *train*). An important property at this level is: *trains cannot overtake*, which is why we introduce the relative ordering of the trains, represented by the variable association *next_train* between instances of class *train*. Therefore, a train can only exit an HLIII area if there is no train immediately in front: this is represented by the guard $tr \notin dom(next\_train)$ added to method ENV_exit_HL3_area of class *train*.

Trackside and Train Movement: In the context, we model the network topology using iUML-B class diagrams (Fig. 2). First we introduce the *TRAIN* class (not shown in figures), then the *VSS* with their linear layout enforced by indexing via attribute *VSS_i*, Fig. 2a.

In the next refinement, we model train movement. A train's position is given by the range of VSS that it occupies: variable association *occupiedBy* in Fig. 1. We only model trains moving forward, hence a train can only leave a VSS if it occupies the next one. In order



(a) VSS Properties



(b) TTD Properties

Fig. 2: Class Diagram representing the track in the context

to ensure the no overtaking property, a train can only move forward if it does not share a VSS with its next train. Apart from splitting, a train can only enter a HLIII area from the first VSS and trains can only leave from the last VSS. Since the no-overtaking property is fundamental to the safety of the system, we ensure the model does not break it by introducing the following invariant, which states that a train cannot occupy a VSS with an index higher than the lowest indexed VSS of the next train:[2]

$$\forall tr1, tr2 \cdot (tr2 \mapsto tr1) \in next\_train \implies max(VSS\_i[occupiedBy \sim [\{tr2\}]]) \leq min(VSS\_i[occupiedBy \sim [\{tr1\}]])$$

To distinguish between trains that are communicating and those that are not, we introduce sub-states *connected* and *ghost*, of *train* (Fig.3).

Next, we introduce the *TTD* which groups sets of contiguous *VSS* via association *Sections* (Fig. 2b). Class *occupiedTTD*, which is a sub-class of *TTD*, represents those TTD that have at least one of their VSS occupied by a train. At this level, we distinguish two cases when a train is leaving the last VSS of the TTD: i) no other train occupies the TTD and the TTD becomes free (and is removed from *occupiedTTD*) or ii) it remains occupied and not free. The same applies to a train exiting a HLIII area which can also free a TTD.

[2] Where $\sim$ is an inverse relation and [] are relational image. A concise summary of Event-B syntax is available at http://wiki.event-b.org/images/EventB-Summary.pdf.

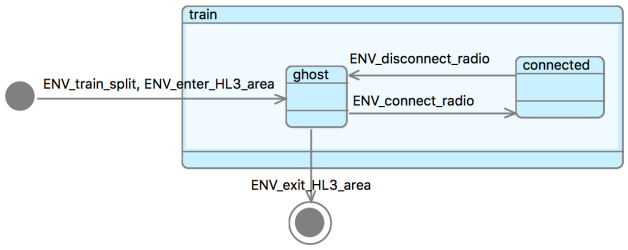


Fig. 3: Train communication state-machine

Movement Authority: In the final environment model, we introduce the RBC role which paves the way for the VBD part. The RBC provides movement authorities (MA) which we assume trains will respect. The MA is modelled as a variable association *ma* between *train* and *VSS*. We refine the train state-machine further by introducing a parallel state-machine (Fig. 4). The sub-states, *inMission* and *noMission*, distinguish the mission status of trains. *inMission* represents trains that have performed a *Start of Mission* (SoM) (transition ENV_start_of_mission), while *noMission* represents trains that either did not start, or performed an *End of Mission* (EoM) (transition ENV_end_of_mission). The mission state-machine was introduced as a parallel state-machine to the communication state-machine so that trains that lose communication retain their mission status. All connected trains have a mission. This is ensured by the invariant: $connected \subseteq inMission$

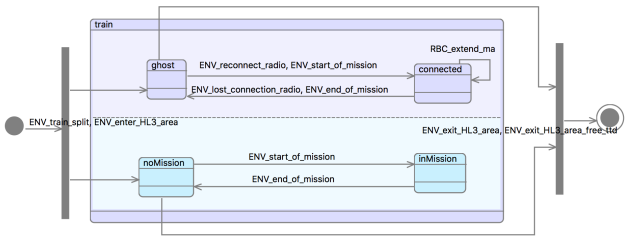


Fig. 4: Parallel state-machines for communication and movement authority

We also split each of the the radio connection/disconnection transitions in Fig. 3 into two cases to distinguish between SoM and reconnection vs connection loss and EoM. The transitions ENV_start_of_mission and ENV_end_of_mission are common to both state-machines. Note that when a train first enters a HLIII area, it enters as a ghost train with no mission, and when leaving the HLIII area it also has to exit as a ghost train with no mission.

When a train performs SoM, it is immediately granted an MA for the VSS it occupies. However, this does not

allow the train to move to new VSS sections. In order to move forward, the RBC should extend the MA as shown by the self transition RBC_extend_ma of the *connected* state in Fig. 4. Our assumption that trains with a mission respect their MA is enforced by the *inMission* class invariant: $occupiedBy \sim [\{tr\}] \subseteq ma[\{tr\}]$ [3].

In [16], we discussed how revoking *ma* (e.g. due to propagation of an unknown VSS state) can result in an unsafe state if the actual train position has progressed sufficiently to occupy the revoked part of the *ma*. In the updated version of the specification [1], the notion of revoking *ma* has been removed; section 4.2.1.6 states that the *ma* will be impacted depending on implementation. It seems there are special measures for these cases to ensure safety, that are not part of this specification document. We therefore simplified our model by removing the RBC_shrink_ma event and the associated *unsafe* attribute flag described in [16] which was designed to highlight the unsafe area of operation. We now verify that safety is ensured by *ma* which is never revoked. In Fig. 1, RBC_trim_ma in the *connected* class plays the role of a garbage collector, removing the VSS the train has left behind.

4.2.2 Modelling the VBD

The VBD cannot see directly what is happening in the ENV; it depends on periodic reports (PTD) sent by the train, and it then asynchronously updates the VSS states. Similarly, the RBC receives information about VSS state from the VBD. This asynchronous behaviour relies on the fact that the actual train position cannot be behind that last reported and is somewhere within the MA. That is, the reported position is only used to free VSS after a train has passed. This is embodied in the following invariants of class *connected* which relate the actual position *occupiedBy* with the *reportedPosition* seen by the VBD.

$$\textbf{inv1: } min(VSS\_i[reportedPosition[\{tr\}]]) \leq min(VSS\_i[occupiedBy \sim [\{tr\}]])$$

$$\textbf{inv2: } max(VSS\_i[reportedPosition[\{tr\}]]) \leq max(VSS\_i[occupiedBy \sim [\{tr\}]])$$

In [16], these invariants were part of class *inMission*, but the new version of the specification introduces memorised location, which is the last location known to the VBD, before the train lost its connection or ended its mission. Fig. 1 presents the association *memorisedLoc* between *ghost* and *VSS*. The same invariants, *inv1* and *inv2*, apply to trains with a memorised location.

Movement Authority & VSS Availability: In the next refinement of the VBD, we distinguish between the two different modes of MA: FSMA and OSMA. In FSMA mode the RBC only uses free VSS to extend *ma*. In OSMA mode, the RBC can extend *ma* with any VSS since we trust the OSMA trains not to crash into the rear of the next train. This behaviour is modelled in Fig. 5 by partitioning *inMission* into two different sub-states, *controlled* and *trusted* representing FSMA and OSMA modes respectively. The choice between the two transitions, RBC_extend_os_ma and RBC_extend_fs_ma, is non-deterministic and determines the mode of the train.

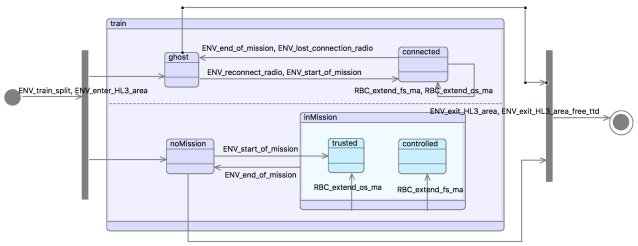


Fig. 5: Introducing sub-states to represent FSMA and OSMA modes

Note that when a train performs a SoM, it is granted OSMA (*trusted*). This is a design decision we took in [16], to ensure safety, since a train is initially not connected when it first enters a HLIII area. In the new version [1], a new section (4.2.2.1.2) is added which notes the risk and suggests the use of the *ATAF* method which uses the OSMA profile to start the mission.

We can now introduce a safety invariant concerning the separation of controlled trains; the *ma* of controlled trains do not overlap:

$$\forall tr1, tr2 \cdot tr1 \in controlled \wedge tr2 \in controlled \setminus \{tr1\} \implies ma[\{tr1\}] \cap ma[\{tr2\}] = \varnothing$$

Hence the RBC can only extend the *ma* of *controlled* trains using VSS sections that are free and not part of any *ma*. We introduce a sub-class *availableVSS* of *VSS* to represent the free VSS sections. This will be refined to the VSS state *free* in future refinements as we introduce the state-machine of the specification. However, extending the *ma* for trusted trains does not have these restrictions.

At this level (VBD-M5) we model an abstraction of the VSS state-machine as shown in the top (m5) part of Fig. 6. We add events to add and remove *availableVSS*

[3] Note that class invariants are implicitly quantified over instances of the class, hence the antecedent $\forall tr \cdot tr \in inMission$ is added automatically.

by generalising the conditions that apply for the VSS state transitions. Therefore for adding an *availableVSS*, it either belongs to a free TTD or no train has reported its position in this VSS, while for now the only condition for removing a VSS from *availableVSS* is that it belongs to an occupied TTD. We also add an event to model 'no change in the availability of the VSS'.

The semantics of the VSS state-machine is a form of *run-to-completion* where a trigger event (e.g. receiving new TTD/PTD information or timer expiry) results in all the consequently enabled transitions of the VSS state-machine being fired to update the affected VSS states before another trigger event is considered. Later, we introduce the notion of triggers explicitly as a scheduling mechanism for the events that require running the VSS state-machine.

We use the *all-replicator* of the *Event Refinement Structure* (ERS) [20] approach to model the *run-to-completion* semantics of the state-machine, where the state of all the VSS sections must be updated before completion allows the next cyle of events. The VBD_vss_no_change event introduced in the last refinement simplifies *completion* by removing the need to calculate what needs to be completed.

Integrity & Position Reports: Next we introduce the concept of train integrity. We partition *connected* into two sub-states: *integral* and *nonIntegral*. We also refine the PTD position reports to include integrity information. Therefore we split the method VBD_receive_position_report into different cases for confirming integrity, integrity loss, integrity not available and train length change. The difference between the position report events is that when integrity is confirmed, both front and rear positions are updated. However, when integrity is not confirmed, only the front position of the train is updated and in later refinements this will have an effect on the integrity timers.

VSS State-machine & Timing: We introduce timing at VBD-M7, by introducing *TIMER_STATUS* which can be either *Idle*, *Started* or *Expired*. We model time abstractly with an ordinal scale, non-deterministically allowing running timers an opportunity to expire. In later refinement, the expired timers will trigger the VSS state-machine to run. The timers are initially *Idle* and some events or transitions will result in changing the timer state to *Started*. For example, a train mute timer will be started when a position report is received. At this level we introduce all waiting timers except the shadow train timer B which depends on VSS sub-states yet to be introduced. The shadow train timer B, similar to shadow train timer A (*inv3*), is assigned to each TTD to mitigate the risk of a shadow train following an integral train. The timers introduced at VBD-M7 are defined as shown below, where the last two invariants represent relationships between the state of timers and the *train* state.

```
1 @inv1: muteTimer ∈ train→ TIMER_STATUS
2 @inv2: waitIntegrityTimer ∈ train→ TIMER_STATUS
3 @inv3: shadowTrainTimerA ∈ TTD →TIMER_STATUS
4 @inv4: ∀tr·tr ∈ dom(memorisedLoc) ⇒muteTimer(tr) = Expired
       ∨tr ∈ noMission
5 @inv5: ∀tr·tr ∈ train ∧waitIntegrityTimer(tr) = Expired ⇒tr
        ∈ nonIntegral
```

The mute and wait integrity timers (*inv1* & *inv2*) are total functions from *train* to *TIMER_STATUS*, so are assigned to every train entering a HLIII area. In the case of *ghost* trains that never establish a connection with the trackside, these timers will remain *Idle* and will never be started. While the shadow train timer A (*inv3*) is a total function from TTD to *TIMER_STATUS*, hence it is associated to every TTD section.

Invariant *inv4* ensures that a train with a memorised location is either a train with expired mute timer (i.e. a train that lost its connection with trackside) or a train that ended its mission. Invariant *inv5* ensures that the state of a train with expired wait integrity timer is *nonIntegral*.

At the next VBD-M8 level, we introduce the full VSS state-machine from the specification document, Fig. 7. The *free* state refines the previous *availableVSS* and the three states *occupied*, *unknown*, *ambiguous* correspond to non-available VSS sections. Therefore, all the transitions going to the *free* state will refine VBD_add_available_vss, and all the transitions leaving the *free* state will refine VBD_remove_available_vss, and finally the remaining transitions will refine VBD_vss_no_change.

In Fig. 7, we model the abstract 12 transitions of the VSS state-machine with additional self-loops for each of the four states. The numbers in front of the transition names correspond to the transition numbers shown in **m8** of Fig. 6 and the transitions described in the specification document.

The guards for self-transitions are the conjunction of the negated guards of all other transitions from the same source state. In other words the self-transition only enables when no other transition is enabled to leave that state. The grey **m8** part of Fig. 6 demonstrates how the events VBD_add_available_vss, VBD_remove_available_vss and VBD_vss_no_change are decomposed to the VSS state transitions.

We also introduce the concept of triggers, which explicitly initiate the running of the VSS state-machine. The presence of a trigger is a guard for VBD_start_vss_update in Fig. 6. We extend the context

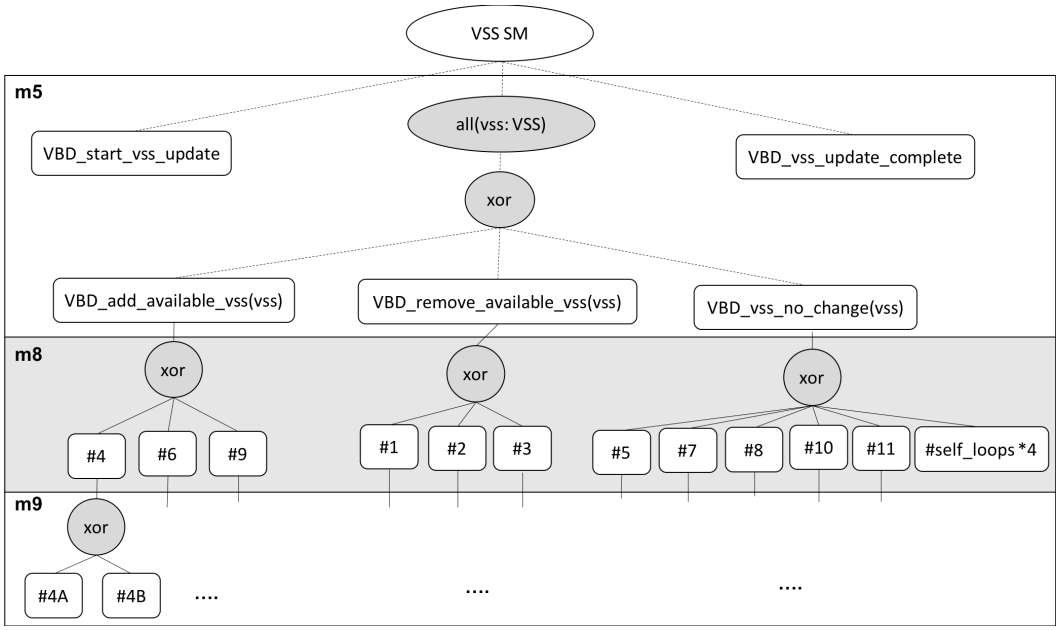


Fig. 6: ERS representation of the VSS state-machine

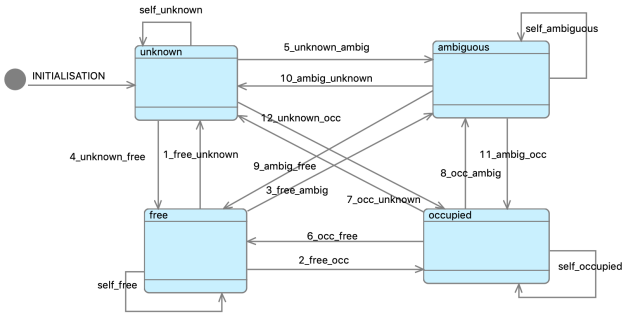


Fig. 7: The VSS state-machine in iUML-B

to distinguish between PTD, TTD, timer expiry and startup triggers. (The startup trigger is only used for the initial state to update the VSS sections starting initially as unknown). The state-machine cannot start dealing with another trigger until it completes updating all the VSS states. To simplify the model we only allow one trigger at a time. This has the disadvantage of always giving some triggers priority over others. In our case TTD triggers have a higher priority. We also introduce the propagation timers and the remaining waiting timer. When the state-machine completes the VSS state update (VBD_vss_update_complete), it will reset the trigger and the propagation timers.

In the last VBD-M9 refinement, we decompose the state-machine transitions to model the different alternatives as presented in the specification table of the VSS state-machine, and add all the remaining details.

5 Verification & Validation

In our modelling approach, we start by verifying safety at the earlier stages of the model and, then later focus on validating the VBD system. We prove safety using the MA concepts, where in [16], we show that revoking MA can result in an unsafe state. However, in the latest version of the specification [1], the notion of revoking MA is removed. Therefore, we remove the RBC_shrink_ma event from our model. However, there remain questions about how MA can be impacted and the conditions that can result in its change. MA is an essential part of the system and to prove safety of the model, more details should be provided.

After verifying system safety, we focus on introducing the details that help us in modelling the VBD state-machine, which is the main concept in the specification document. The table describing the state transitions of the VBD state-machine is very difficult to follow, hence the provided operational scenarios play a major role in validating the model. For validation, we use ProB and BMotionStudio to run the scenarios. We use BMotionStudio to display the main variables in a tabular format. This provides the criteria for validating the scenarios. For example, we present the VSS state, the actual position of the trains and their reported positions, as shown by the upper left part of Fig. 8. We have also used the *CODA Oracle Simulator* [21] to record and replay the event traces when running the scenarios. This is shown in the lower left part of Fig. 8.

5.1 Verification

Most of the manually verified proofs are related to invariants involving *min* and *max* where we used case distinctions to consider boundary cases. Take, for example, the proof of the *no-overtaking* invariant for ENV_front_move_next in ENV-M0. This proof requires the consideration of several cases: the train is the following train (tr_p), the train is the followed train (tr_n), further sub-cases such as the VSS is in the *occupiedBy* etc. In the next iteration of the model, we will look at improving the degree of automation by introducing additional theorems.

When defining the variable types, we try to apply total functions instead of partial functions as much as possible e.g. *reportedPosition* and timers, this improves discovering missing guards and actions. Moreover, we 'lift' the state-machine to a set of instances (*train*). Therefore, the generated state-machine type invariants are based on subsets of the instance set (*train*), which makes it easier to define the whole domain of the functions.

5.1.1 Proof Statistics

In Table 1, we present the proof statistics of our model. Our modelling resulted in 621 proof obligations. The number that are discharged automatically varies depending on the proof tactic profile used and the machine. Our best results were obtained using a proof tactic profile that includes the 'relevance filter' RF (using ML and PP from the AtelierB provers) and then runs the SMT solvers (CVC4 and Z3). The relevance filter is a meta prover that improves the efficiency of the selected theorem provers by localising relevant theorems. In the best case we achieved 95% automatic proof but this dropped to about 88% on slower machines. For comparison, the best results using the default prover configuration of Rodin (after installing AtelierB provers) was 47%.

Table 1: Proof Statistics: Customised Auto-Tactics

Name	Total	Auto	Manual	Und.
ENV_C00	0	0	0	0
ENV_C0	7	7	0	0
ENV_C1	2	0	2	0
ENV_C3	0	0	0	0
VBD_C7	0	0	0	0
VBD_C8	0	0	0	0
ENV_M00	4	4	0	0
ENV_M0	29	26	3	0
ENV_M1	18	18	0	0
ENV_M2	11	11	0	0
ENV_M3	64	62	2	0
VBD_M4	107	95	12	0
VBD_M5	32	32	0	0
VBD_M6	60	60	0	0
VBD_M7	66	66	0	0
VBD_M8	139	137	2	0
VBD_M9	82	75	7	0
Total	**621**	**593**	**28**	**0**

5.2 Validation: Role of ERTMS Operational Scenarios

When validating the scenarios, we instantiated the model with the example given in the scenario to be able to animate and compare the results. In some cases, we have recorded a partial pass. This means we detected some differences, but managed later to synchronise and get the same results. Below we describe the results of running the scenarios and the corresponding changes to the model.

Scen. 1: A normal scenario for integrity confirmation. Pass after updating #6A guards as a result of running step 4. In the specification, #6A states a VSS changes from occupied to free if an integral train leaves the evaluated VSS. We added to the Event-B model an additional condition to take into account a free TTD.

Scen. 2: Train splitting with confirmed integrity. Pass after updating #11A as a result of running step 7. We add a guard to transition #11A that takes into consideration that an integral train has left the rear TTD. This condition is explicitly mentioned in the reference section 4.5.1.4 of the specification.

Scen. 3: Shadow train scenario. Partial pass, we do not model travel distance, hence shadow train timer B is started resulting in enabling #11B in step 7.

Scen. 4: Start and end of mission. Pass, similar to scenario 2 because it runs steps 6 to 8 of scenario 2.

Scen. 5: Integrity loss. Partial pass, we have different states in steps 3 and 7. The main reason of the discrepancy is that we do not model delay in TTD communication and always give priority to TTD triggers over PTD.

Scen. 6: Train disconnecting and reconnecting. Partial pass, resulted in updating #1B, changing the state of a VSS from *free* to *unknown*, to start the ghost propagation timer.

Scen. 7: Connection loss and reconnection with VSS release. Partial pass, resulted in updating transitions related to #8 (changing the state from *occupied* to *ambiguous*), by adding the guard that TTD is occupied. The additional condition is not described in the specification table. Moreover fixing the pri-

ority guards of #12 (*unknown* to *occupied*) over #5(*unknown* to *ambiguous*). We model transition priorities by negating the guards of the higher priority transition. Discrepancies in step 6 because we do not run the state-machine twice and separate the the update of the front and rear positions of the train.

Scen. 8: Sweeping the track and the jumping train effect. Fail, step 4 demonstrates the need to separate front and rear position updates by running the state-machine twice.

Scen. 9: Ghost train. Partial pass, requires strengthening self transitions to make the state-machine deterministic, this is left for the next iteration when an automatic tool is developed to generate the run-to-completion conditions. For now we always give normal transitions a priority over self-transitions when running the scenarios.

Looking at the results, our model fails in the case of jumping trains. Jumping trains, as defined by [1], are trains that the trackside cannot locate due to the discrete position reports and/or delay in trackside detection. Jumping trains are not mentioned much in the specification and the solution of how to avoid this effect is not very well explained. The problem is we do not run our VBD state-machine twice to separately update the front and the rear positions. During our validation phase, we discovered that this is important to avoid the jumping train effect but it is not explained in the later version of the specification [1]. In our current solution, in the case where integrity is not confirmed, we update the VBD position in one step by prepending the new position to the front of the current position. We assumed this would be equivalent to running the state-machine twice, but it does not eliminate the jumping trains effect. Listings 1 and 2 show the difference between assigning *reportedPosition* (@act1) in the events VBD_receive_position_report_integrity_confirmed and VBD_receive_position_report_integrity_loss. In the case where integrity is confirmed, we set the train position (*reportedPosition*) to be the reported position (*pos*) of the train; hence the rear is updated as well as the front. However in the case of integrity loss, the train position becomes the union of the current position (*reportedPosition*) and the reported position (*pos_new*) so that the rear is not updated. The witness (keyword *with* in listing 2) gives the refinement relation between the new and old parameters. The rest of the guards are needed to determine the possible position values and the other actions to set the state of the train as integral or not.

```
event VBD_receive_position_report_integrity_confirmed
refines VBD_receive_position_report
any tr pos
where
@instanceType_tr: tr ∈ connected
@grd1: pos ⊆ma[{tr}]
@grd2: pos ≠∅
@grd3: min(VSS_i[pos]) ≤ min(VSS_i[occupiedBy∼[{tr}]])
@grd4: max(VSS_i[pos]) ≤ max(VSS_i[occupiedBy∼[{tr}]])
@grd5: tr ∉ train_split
@grd_contig: VSS_i[pos] = min(VSS_i[pos]) .. max(VSS_i[pos])
@grd6: min(VSS_i[pos]) ≥ min(VSS_i[reportedPosition[{tr}]])
then
@leave_nonIntegral: nonIntegral := nonIntegral \{tr}
@enter_integral: integral := integral ∪{tr}
@act1: reportedPosition := ({tr} ⩤reportedPosition) ∪({tr}
      × pos)
end
```

Listing 1: Position and Integrity Confirmation Event

```
event VBD_receive_position_report_integrity_loss
refines VBD_receive_position_report
any tr pos_new
where
@instanceType_tr tr ∈ connected
@grd1: pos_new ⊆ma[{tr}]
@grd2: pos_new ≠∅
@grd3: min(VSS_i[pos_new]) ≤ min(VSS_i[occupiedBy∼[{tr}]])
@grd4: max(VSS_i[pos_new]) ≤ max(VSS_i[occupiedBy∼[{tr}]])
@grd5: tr ∈ train_split
@grd_contig: VSS_i[pos_new] = min(VSS_i[pos_new]) .. max(
      VSS_i[pos_new])
@grd6: min(VSS_i[pos_new]) ≥ min(VSS_i[reportedPosition[{tr
      }]])
with
@pos: pos = pos_new ∪reportedPosition[{tr}]
then
@leave_integral: integral := integral \{tr}
@enter_nonIntegral: nonIntegral := nonIntegral ∪{tr}
@act1: reportedPosition := reportedPosition ∪({tr}× pos_new
      )
@act2: train_split := train_split \{tr}
end
```

Listing 2: Position and Integrity Loss Event

In Fig. 8, we show how running step 4 of scenario 8 results in an event error. This problem is captured by the provers, Fig. 9 where we cannot prove the contiguity guard in VBD_receive_position_report_integrity_loss because of a weakened guard between VBD-M5 and VBD-M6 in the case of integrity loss. When decomposing the position reporting event to distinguish between the integrity cases in VBD-M6, we introduce a new parameter *pos_new* which is contiguous but the witness replacing *pos* is not. In this case, running scenario 8 has highlighted a problem in our model, helped us to understand why we could not discharge the proof obligation, and why we actually need to run the state-machine twice to update the VBD position. The solution is to make a clear distinction between three types of positions: the environment position, which we modelled as *occupiedBy*, the position reports and the VBD position (*reportedPosition*) which requires running the state-machine twice in case of PTD and TTD triggers. Since we need to separate the VBD position update into two steps, modelling the position report as a parameter (*pos*) in the position reporting events is not enough, and it needs to be recorded as a variable. This

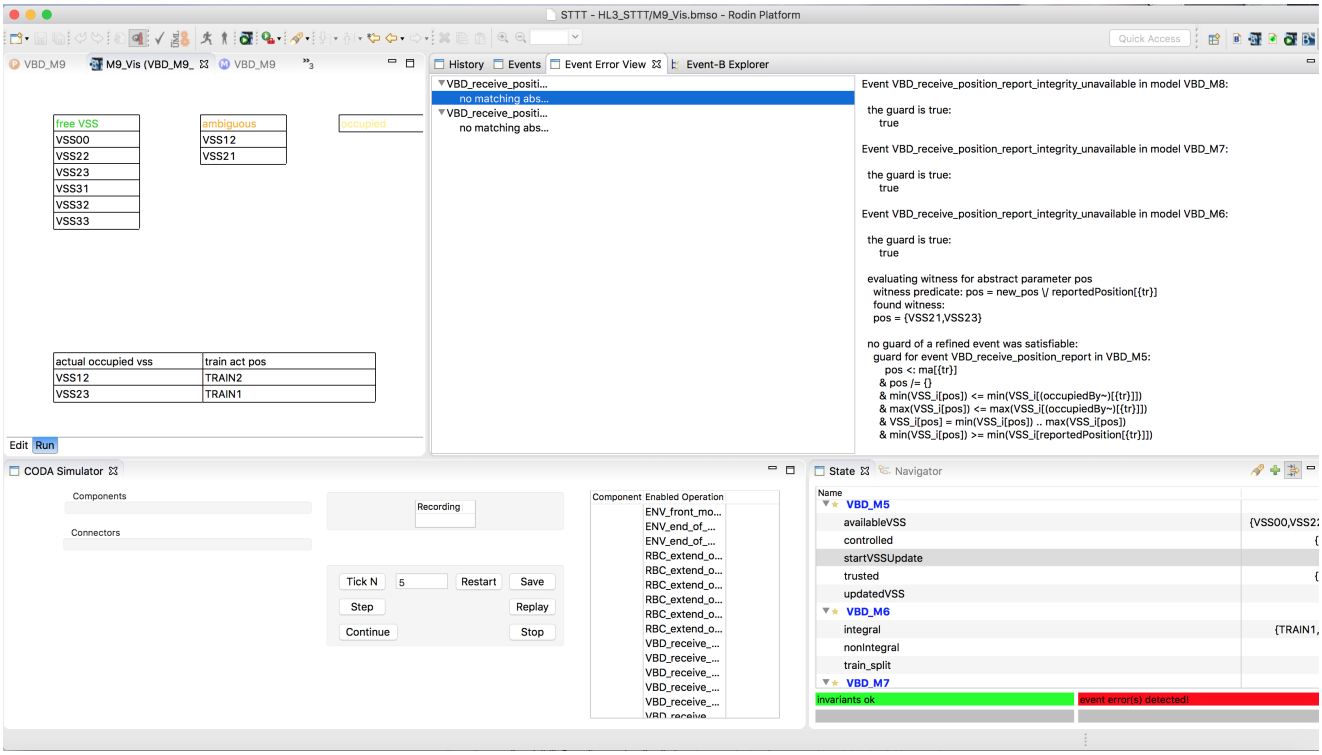


Fig. 8: Validation Example Using ProB and BMotionStudio

Fig. 9: GRD Proof Obligation

update requires another iteration of the model. After this we will be able to prove that the VBD positions are contiguous, but the position reports in comparison to previous reports are not necessarily contiguous (i.e. jumping train effect).

6 Towards Implementation of VBD

We previously mentioned that the focus of this work is on modelling and verification of VBD. However, in order to be able to model and verify different safety requirements such as no-collision, we also had to model other parts of the system, i.e. ENV and RBC. In this section, we briefly explore some of the possible options for generating executable code from our model and introduce a new approach for generating code using an experimental tool.

One of the possible approaches for code generation is to use Tasking Event-B [22] code generation tool. To be able to use Tasking Event-B, we will need to refine the VBD model further until we have concrete data structures and events. Due to this, the first step for using the tool and also tackling the complexity of the model (through separating different components of the system) could be model decomposition [23]. Decomposing VBD from the rest of the model will allow us to focus on this component and refine it towards a concrete level. At the most concrete level, we should provide explicit control flow for the model. The tool is able to generate code for a number of different target languages based on the provided control structure and the model. The tool has been used with Rodin 3.3 for generation of executable code from an Event-B model of an intelligent runtime management software for multi-core embedded platforms [24].

Tasking Event-B has some limitations that makes it difficult to use in some cases. For instance, the tool facilities for defining program structure is restricted (e.g. nested structures are not supported) and only supports concrete structures [24].

Since the gap between the abstract data structures used in our model and concrete data structures that should be used in an efficient implementation is rather large, and due to the aforementioned limitations, Tasking Event-B could not be used until the model is further refined. Based on this, we decided to use a new experimental tool called SEB-CG [25] for providing some suggestions on how we can derive code from our abstract model.

SEB-CG implements the ideas presented in [26] and [27] where Event-B models are augmented with a schedule. The schedule makes the control flow between events explicit from the abstract level by allowing schedule refinement along with the Event-B refinement. Similar to Tasking Event-B, generating executable code with SEB-CG can be done if the model is refined to a concrete level. However, the SEB-CG support for abstract control structures (i.e. iterations and choices) in addition to concrete ones (i.e. loops and branches) and its flexibility in defining nested program structures makes it useful for deriving the algorithmic structure of the model even if the final concrete model is not constructed yet.

Using SEB-CG, we initially introduce an abstract schedule for the most concrete model of VBD:

```
1  schedule code0
2  machine VBD_M9_Anim1
3  proc vbd(in:vss_in)
4  begin
5    VBD_start_vss_update
6    do
7    {
8      1A_free_unknown
9    []
10     1B_free_unknown
11   []
       .
12     .
13     .
14   []
       self_ambiguous
15   []
       self_occupied
16   }
17   od
18   VBD_vss_update_complete
19 end
```

The above schedule has two main purposes: 1) it groups relevant VBD events using the `proc` construct (not all events are shown) and 2) it provides explicit abstract control flow between events. `do..od` is an abstract iteration and { `...` `[]` `...` } is a non-deterministic choice. Elements in a schedule are executed based on the schedule order. We refine the above abstract schedule towards a more concrete one as follows:

```
1  schedule code refines code0
2  machine VBD_M9_Anim1
3  proc vbd(in:vss_in)
4  begin
5    VBD_start_vss_update
6    do
7    if(vss_in ∈ free  ∧  vss_in ∉ updatedVSS  ∧
         trigger = TTDInfo  ∧
         (∃ttd.ttd = Sections^−1(vss_in)  ∧  ttd ∈ occupiedTTD
           ∧  Sections[{ttd}] ∩ ma[controlled] = ∅
           ∧  Sections[{ttd}] ∪ ran(reportedPosition) = ∅))
8      {
9        1A_free_unknown
10     }
11   elseif(vss_in ∈ free  ∧  vss_in ∉ updatedVSS  ∧
         vss_in ∉ ran(reportedPosition)  ∧
         (∃vss_prop.vss_prop ≠ vss_in  ∧
         Sections^−1(vss_prop) = Sections^−1(vss_in)  ∧
         disconnectPropagationTimer(vss_prop) = Expired
           ∧  (∀v.VSS_i(v) > min(VSS_i[{vss_prop, vss_in}])
           ∧  VSS_i(v) < max(VSS_i[{vss_prop, vss_in}]) ⟹
         v ∈ (free ∪ unknown)))  ∧
         vss_in ∉ ran(reportedPosition))
12     {
13       1B_free_unknown
14     }
       .
15     .
16   od
17   VBD_vss_update_complete
18 end
```

The above schedule refinement refines the abstract choice given in schedule `code0` to concrete `if..else` branches. The explicit branch conditions are extracted from the respective event guards. Since the model is still abstract, some of the computations required for deciding which case should be executed are specified using event parameters and guards. At this stage, we translate those parameters and their guards to an existential quantifier. These existential quantifications can be seen as the specification of other procedures, so if there is a procedure that satisfies this existential quantification then the quantification in the branch condition can be replaced by a procedure call to that procedure (which should return a boolean value) in code level. Using an approach similar to [28] it is possible to transform the guards to a set of pre- and post-conditions in a language which is supported by a static verifier and then implement and verify the aforementioned procedures there.

Using the above approach, we can gain insight into the structure of the final implementation even though the model is not refined to a concrete level yet. The difficulty of refining the Event-B model of a complex example, such as the one in this paper, to a level where all the event parameters and abstract data structures can be replaced by concrete variables, suggests the need for a combinational approach where high level properties are specified and verified in Event-B and low level code-oriented properties are verified using a program verifier.

7 Other Observations

In this section we identify some ambiguities and limitations in the HLIII specification document and suggest improvements. We also suggest some improvements

to the tools based on our experience of modelling the HLIII, some of which are already in progress.

Specifications: The specification document [1] mentions several times, "while the MA is still valid", however the document does not explain when an MA becomes invalid. As we have shown in [16], revoking an MA can lead to an unsafe state. Similarly, the specification mentions that the MA can be "impacted", but does not explain how. In some scenarios (e.g. scenario 6), the MA has changed between steps. We assume this is a mistake in the specification since otherwise the MA appears to have been revoked. As mentioned earlier in Sec. 5.2, the rationale for running the VSS state-machine twice in the case of TTD and PTD trigger events, should have been explained and emphasised more. This led to us making incorrect assumptions about the behaviour of our model. In the VSS state-machine transitions table [1], the condition for #6A is "integral train has left the evaluated VSS". When modelling the conditions for the transitions, we missed the case where TTD is free. Although this is covered by an integral train leaving the VSS, it was only discovered by running the scenarios. It would be better to present this condition clearly as a separate transition (i.e. introduce new condition #6B: TTD is free). Moreover, transition #7A changing the state of a VSS from *occupied* to *unknown*, if the evaluated VSS is part of a train memorised location with an expired mute timer or no communication session, has a priority over #6A which changes the state of the VSS from *occupied* to *free*. Does this priority still hold if TTD is free?

In scenario 6, which covers the case of a train losing its radio communication and later reconnecting, there is a discrepancy with Sec. 3.4.2.2.2 in p.16 of [1], which describes the change of the VSS state-machine to free as one of the stopping events for the disconnect propagation timer. On the other hand, in steps 4 & 5, when *vss12* becomes free the timer should be stopped according to p.16. Hence it cannot expire in step 5. The scenario also did not mention the start of the propagation timer for *vss21* and *vss22*, because both sections are part of a train MA for which the mute timer has expired. In this case one of these timers can expire and the transition should be #1C instead of #1D in step 5 of scenario 6, i.e. the VSS state will become *unknown* due to the expiry of a propagation timer on the same TTD (#1C) rather than a different TTD (#1D).

Tooling: The iUML-B diagrammatic notation, helped us to express and communicate the models between the team members and followed naturally from our system analysis and review meetings. In later stages when substantial reworking of the model was required, the diagrams somewhat hindered progress because changes often have to be repeated throughout the refinement chain. As a result we intend to improve the refactoring features of iUML-B tooling.

```
statemachine SM(variables) // Statemachine SM with "variables" translation

    Initial Init1 // An initial state Init1
    State S0 // A state S0
            statemachines
            begin
                statemachine SM1(variables) // Nested statemachine SM1
                    State T0
                    State T1
            end
        State S1

    /* Below are the transitions */
    transition
        elaborates INITIALISATION
        target "S0.SM1.T0"
        source Init1
    end
```

Fig. 10: Textual Representation of iUML-B

Modelling a specification such as HLIII involves iterative review and refactoring over refinements which is made more cumbersome by having to re-draw diagrams. Textual representations can be more efficient to refactor using simple text copy and paste operations. We are currently developing a textual representation of iUML-B using Xtext [29], which is also beneficial for tracking changes and supporting version control. Fig. 10 shows a snippet of a textual representation of a state-machine.

It would have been useful to be able to structure the model into components for VBD, RBC and ENV. Model composition tools, based on inclusion of machines [30], are available but are not yet compatible with iUML-B. We are developing a containment mechanism so that iUML-B can be used with inclusion. In future work we will re-structure the model to assess these composition techniques.

Our experiences of replaying large and detailed scenarios gave us strong motivation to improve tool support for running scenarios. The *CODA Oracle Simulator* has some useful features but still requires each event of the system and controller to be manually selected in the correct sequence. When replaying scenarios any differences in state or enabledness due to a change in the model are discovered and halt the reply. This could be improved if the user was able to configure which are the important observable variables and events that should match the recording and which are internal detail that may be allowed to vary.

8 Comparison

Our work on the HLIII specification began in the case study track of the ABZ 2018 conference. In this section we compare our approach [16] with the six other contributions to the ABZ 2018 Case study track.

Three of these approaches [31,32,33] are based on Event-B and use theorem proving to verify the main principles behind the specification irrespective of scenarios. These contributions, like ours, are based on abstraction and refinement rather than the operational details of the VBD scenarios. Of these theorem proving contributions, Mammar et al. [33] covered the most detail including the VBD state-machine behaviour, albeit in a rather cumbersome last refinement with a single update event. Fotso et al. [32] use a goal structured analysis (KAOS) to drive the refinement structure but do not model the closed system (i.e. including environment). Abrial's Event-B contribution [31] focuses on synthesising a clearer statement of the requirements and consequently does not progress beyond the first abstract model. The model does however include a substantial, though simplified, subset of the requirements which prove the main principle of the VBD. We agree with Abrial that the specification is operational (or analytic) in nature. Hence our systems analysis to extract the requirements of the VBD subsystem and our reports of increasing our understanding through iterative modelling attempts.

The remaining three contributions do not attempt any abstraction but model the concrete specification as faithfully as possible in order to model check or animate it to discover bugs against the scenarios. Cunha et al. [34] do this using Electum (an extension of Alloy) and the Analyzer model checker, and Arcaini et al. [35] use Promela with the Spin model checker. Hanson et al. use ProB to execute 'classical' B [36] to demonstrate the specification controlling an actual (test) railway system. This contribution does not need to model the environment since it validates via 'Model-in-the-loop'.

Although our ABZ contribution lacked the VBD state-machine behaviour needed to validate scenarios, we have now 'caught up' by making further refinements to model the state-machine via a triggering process. In this paper we attempt to achieve the best of both: the proof of the abstract principles of safe operation and the validation of the operational specification against its scenarios.

Train control is a familiar domain for Formal Methods, and specifically for B and Event-B-based approaches. Butler et al [37] give a methodical treatment of the diagrammatic modelling of the rail interlocking system Railground with both iUML-B and Event Refinement Structures [38]. In [19], the authors present the Event-B development of a *Communications-based Train Control* (CBTC) system from Hitachi Ltd. Their focus is on the use of *Abstract Data Types* (ADTs) to manage the complexity of modelling a graph-based rail network and its dynamics. This example is comparable to *European Rail Traffic Management System* (ERTMS) Level 3 and uses moving blocks. The authors further proposed [39] the extension of iUML-B to support diagrammatic modelling of ADTs, using the same Railground case study as [37].

Other related work such as [40] on Hybrid ERTMS Level 3 is based on moving blocks. These models are hybrid, being concerned with continuous modelling of exact train position and speed reporting. This ABZ2018 case study is the first formal examination of fixed virtual blocks that we are aware of.

9 Conclusions

To summarise, we have performed a full formal development involving the following:

- Systems analysis to synthesise requirements from a detailed operational specification.
- Iterative formal modelling to develop our understanding of the requirements.
- Abstraction of the environment and important safety properties as a formal model.
- Refinement to introduce an abstract model of the VBD control component.
- Refinement to introduce operational details of the VBD control component.
- Use of diagrammatic modelling notations to increase understanding and structuring of the models.
- Validation of the models by animation of the given scenarios.
- Preliminary work towards generating an implementation.

The result is a formal model of the VBD specification that is proven to be safe (with some caveats that still need clarification) and has been demonstrated to accurately represent the specification's behaviour.

Our formal verification using theorem provers, of the safety of the HLIII specification has been extremely beneficial in identifying potential problem areas. While industry experts are aware of the engineering decisions behind the specification, unverifiable safety requirements have encouraged re-consideration of some critical behaviours in the specification. To influence a major European standard is a significant achievement. Our model captures the abstract principles behind HLIII and re-

fines them with the full operational details of the specified state-machine behaviour.

For validation, we used a 'scenario checker' plug-in tool to enhance ProB animation and BMotionStudio to visualise the state being checked. We discovered one area in the scenarios, where our model deviates from the expected behaviour. The failure of the 'jumping trains' scenario requires a new iteration of the model, where we will mainly focus on updating the VBD front and rear position separately.

A List of Abbreviations

ADT	Abstract Data Type
Rodin	Rodin platform
ERS	Event Refinement Structure
CBTC	Communications-based Train Control
ERTMS	European Rail Traffic Management System
HLIII	Hybrid ERTMS Level 3
VBD	Virtual Block Detector
ENV	environment
RBC	Radio Block Centre
VSS	Virtual Sub-Section
TTD	Trackside Train Detection
PTD	Positive Train Detection
SoM	Start of Mission
EoM	End of Mission
TIMS	Train Integrity Monitoring System
MA	Movement Authority
FSMA	Full Supervision Movement Authority
OSMA	On-Sight Movement Authority
PO	Proof Obligation

Acknowledgements This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Union's HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

References

1. EEIG ERTMS Users Group. *Principles: Hybrid ERTMS/ETCS Level 3*, 1c edition, July 2018. https://ertms.be/sites/default/files/2018-07/16E0421C_HL3-clean.pdf. Accessed 24/1/2019.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. Atif Mashkoor, Felix Kossak, and Alexander Egyed. Evaluating the suitability of state-based formal methods for industrial deployment. *Softw., Pract. Exper.*, 48(12):2350–2379, 2018.
4. Egon Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, Heidelberg, 2003.
5. J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
6. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.
7. J. Paul Gibson and Isabelle Perseil. Introduction to UML and Formal Methods. *SIGSOFT Softw. Eng. Notes*, 37(4):32–33, July 2012.
8. Maria Encarnación Beato, Manuel Barrio-Solórzano, Carlos E. Cuesta, and Pablo de la Fuente. Formal methods for UML. In Hossam A. Gabbar, editor, *Modern Formal Methods and Applications*. Springer Netherlands, Dordrecht, 2006.
9. Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.
10. Colin Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, pages 29–30, Toulouse, France, 2014. http://eprints.soton.ac.uk/365301/.
11. Mar Yah Said, Michael Butler, and Colin Snook. A method of refinement in UML-B. *Softw. Syst. Model.*, 14(4):1557–1580, 2015.
12. Felix Kossak, Atif Mashkoor, Verena Geist, and Christa Illibauer. Improving the understandability of formal specifications: An experience report. In *Proceedings of the 20th International Working Conference on Requirements Engineering: Foundation for Software Quality - Volume 8396*, REFSQ 2014, pages 184–199, Berlin, Heidelberg, 2014. Springer-Verlag.
13. Cheng Pang, Antti Pakonen, Igor Buzhinsky, and Valeriy Vyatkin. A study on user-friendly formal specification languages for requirements formalization. In *14th IEEE International Conference on Industrial Informatics, INDIN 2016, Poitiers, France, July 19-21, 2016*, pages 676–682. IEEE, 2016.
14. Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.
15. Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B models with B-Motion Studio. In *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.
16. Dana Dghaym, Michael Poppleton, and Colin Snook. Diagram-led formal modelling using iUML-B for Hybrid ERTMS Level 3. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 338–352, Cham, 2018. Springer International Publishing.
17. EEIG ERTMS Users Group. *Principles: Hybrid ERTMS/ETCS Level 3*. http://www.southampton.ac.uk/assets/sharepoint/groupsite/Academic/ABZ-Coneference-2018/Public%20Documents/ABZ2018/16E0421A_HL3.pdf. Accessed 18/1/2018.
18. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
19. Andreas Fürst, Thai Son Hoang, David Basin, Naoto Sato, and Kunihiko Miyazaki. Large-scale system development using Abstract Data Types and refinement. *Sci. Comput. Program.*, 131:59–75, 2016.

20. Asieh Salehi, Michael Butler, and Abdolbaghi Rezazadeh. Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*, 27(3):499–523, May 2015.
21. Michael Butler, John Colley, Andrew Edmunds, Colin Snook, Neil Evans, Neil Grant, and Helen Marshall. Modelling and refinement in CODA. *Electronic Proceedings in Theoretical Computer Science*, 115:36–51, 05 2013.
22. Andrew Edmunds and Michael Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES2011*, April 2011.
23. Michael Butler. Decomposition structures for Event-B. In *International Conference on Integrated Formal Methods*, pages 20–38. Springer, 2009.
24. Mohammadsadegh Dalvandi, Asieh Salehi Fathabadi, and Michael Butler. A report on PRiME code generation activities. In *7th Rodin Workshop (05/06/18)*, June 2018.
25. Mohammadsadegh Dalvandi, Michael Butler, and Asieh Salehi Fathabadi. SEB-CG: Code Generation Tool with Algorithmic Refinement Support for Event-B. In *Workshop on Practical Formal Verification for Software Dependability (AFFORD'19)*, 2019.
26. Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. Derivation of algorithmic control structures in Event-B refinement. *Science of Computer Programming*, 148:49–65, 2017.
27. Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi. Verifiable code generation from scheduled Event-B models. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 234–248. Springer, 2018.
28. Mohammadsadegh Dalvandi, Michael Butler, and Abdolbaghi Rezazadeh. Transforming Event-B models to Dafny contracts. *Electronic Communications of the EASST*, 72, 2015.
29. Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.
30. Thai Son Hoang, Dana Dghaym, Colin Snook, and Michael Butler. A composition mechanism for refinement-based methods. In *Proceedings 2017 22nd International Conference on Engineering of Complex Computer Systems: ICECCS 2017*. IEEE, February 2018.
31. Jean-Raymond Abrial. The ABZ-2018 case study with Event-B. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 322–337, Cham, 2018. Springer International Publishing.
32. Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, and Amel Mammar. Modeling the Hybrid ERTMS/ETCS Level 3 standard using a formal requirements engineering approach. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 262–276, Cham, 2018. Springer International Publishing.
33. Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso, and Régine Laleau. An Event-B model of the Hybrid ERTMS/ETCS Level 3 standard. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 353–366, Cham, 2018. Springer International Publishing.
34. Alcino Cunha and Nuno Macedo. Validating the Hybrid ERTMS/ETCS Level 3 concept with Electrum. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 307–321, Cham, 2018. Springer International Publishing.
35. Paolo Arcaini, Pavel Ježek, and Jan Kofroň. Modelling the Hybrid ERTMS/ETCS Level 3 case study in SPIN. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 277–291, Cham, 2018. Springer International Publishing.
36. Dominik Hansen, Michael Leuschel, David Schneider, Sebastian Krings, Philipp Körner, Thomas Naulin, Nader Nayeri, and Frank Skowron. Using a formal B model at runtime in a demonstration of the ETCS Hybrid Level 3 concept with real trains. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 292–306, Cham, 2018. Springer International Publishing.
37. Michael Butler, Dana Dghaym, Tomas Fischer, Thai Son Hoang, Klaus Reichl, Colin Snook, and Peter Tummeltshammer. Formal modelling techniques for efficient development of railway control products. In *RSSRail 2017, Pistoia, Italy, 2017*, volume 10598 of *LNCS*, pages 71–86. Springer, 2017.
38. Asieh Salehi, Michael Butler, and Abdolbaghi Rezazadeh. Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*, 27(3):499–523, 2015.
39. Thai Son Hoang, Colin Snook, Dana Dghaym, and Michael Butler. Class-diagrams for Abstract Data Types. In *ICTAC 2017, Hanoi, Vietnam, 2017, Proceedings*, volume 10580 of *LNCS*, pages 100–117. Springer, 2017.
40. André Platzer and Jan-David Quesel. European Train Control System: A case study in formal verification. In *ICFEM 2009, Rio de Janeiro, Brazil, 2009. Proceedings*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.